
madsage
IRan Education
Research
NETwork
(IRERNET)
شبکه آموزشی-پژوهشی مادسیج
با هدف بهبود پیشرفت علمی
و دسترسی راحت به اطلاعات
برای جامعه بزرگ علمی ایران
ایجاد شده است
مادسیج
شبکه آموزشی -پژوهشی ایران
madsg.com
مادسیج

فصل 8

اشاره‌گرها

هدف کلی
آشنایی با کاربردهاي متعدد اشاره‌گرها در زبان C

هدفهای رفتاری
از دانشجو انتظار می‌رود پس از مطالعة این فصل،
1. با تعریف و کاربردهای اشاره‌گرها آشنا شود.
2. با نحوة معرفی اشاره‌گرها در برنامه آشنا شود.
3. کاربرد اپراتور یکانی & و عملگر * را بشناسد.
4. با نحوة آدرس‌دهی داده‌ها آشنا شود.
5. نحوة مقداردهی اولیه به اشاره گرها را بداند.
6. کاربرد اشاره‌گر تهی را بشناسد.
7. با سه عملیات انتساب، محاسبه و مقایسه بر روی اشاره‌گرها آشنا شود.
8. نحوة گذردادن آرگومانها با آدرس و با مرجع را بداند.
9. با کاربرد انتقال دوطرفة اطلاعات در اشاره‌گرها آشنا شود.
10. رابطة بین اشاره‌گرها و آرایه‌های تک‌بعدی و دوبعدی را بداند.
11. با تعریف آرایه به صورت قراردادی آشنا شود.
12. با نحوة انتقال آرایه به تابع آشنا شود.
13. نحوة تعریف آرایه‌ای از اشاره‌گرها را بداند.
14. مفهوم اشاره‌گر به اشاره‌گر با بشناسد.
15. نحوة ارسال اشاره‌گرها به آرگومان به توابع را بداند.

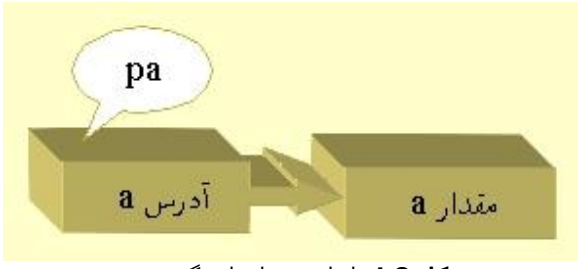
مقدمه
در اغلب زبانهاي برنامه نويسي قديمي، مانند فورترن و كوبول ، مفهومي به نام اشاره‌گر وجود ندارد . اما يكي از ويژگيهاي بارز زبان C، كاربرد متعدد اشاره‌گرها و انجام عمليات محاسباتي روي آنهاست. اشاره‌گر متغيري است كه آدرس متغير ديگري را در خود نگه مي دارد؛ يعني به آدرس متغير ديگر اشاره مي كند. به عبارت ديگر مقدار آن، آدرس يك خانه از حافظه است . اشاره‌گر روش غيرمستقيم دسترسي به داده هاست و كاربردهاي زيادي در C دارد كه از آن جمله مي‌توان موارد زير را عنوان كرد.
- انتقال آدرس متغيرها به تابع فرعي
- برگرداندن چندين مقدار از تابع فرعي
- دستيابي به عناصر آرايه‌ها
- تشكيل ساختارهاي پيچيده‌تر مانند فهرستهاي پيوندي، درختها و نمودارها
- عمل تخصيص حافظه به صورت پويا.

نحوة معرفی اشاره‌گر
براي استفاده از اشاره‌گر در برنامه، ابتدا بايد اشاره‌گر تعريف شود. روش كلي تعريف متغيري از نوع اشاره‌گر به صورت زير است.
data-type * ptvar ;
كه در آن ptvar نام متغير مورد نظر و data-type نوع متغيري است كه آدرس آن در متغير اشاره‌گر ptvar قرار می‌گيرد. نماد `*´ نيز اپراتور اشاره‌گر است.

متغيرهاي اشاره‌گر ممكن است به متغيرهاي عددي، كاراكتري، آرايه ها، توابع، ساختارها يا ديگر متغيرهاي اشاره‌گر اشاره كند. در حالت كلي هر نوع داده ذخيره شده در حافظ ة كامپيوتر يك يا چند بايت متوالي از خانه هاي حافظه را اشغال مي كند. در صورتي مي توان به داده دسترسي داشت كه آدرس اولين خانه يا اولين بايت آن را در حافظه بدانيم . آدرس محل متغير a در حافظه با عبارت &a تعيين مي‌گردد كه در آن & اپراتور يكاني يا تك‌اپراندي است و اپراتور آدرس ناميده مي‌شود و آدرس اپراند يا عملوند خود را به دست مي‌دهد. حال فرض كنيد كه متغير a از نوع int و متغير pa نيز متغير اشاره‌گر باشد و به صورت زير توصيف كرده باشيم.

int *pa ;

در اين صورت با دستور جايگذاري pa = &a ; آدرس متغير a به اشاره‌گر pa نسبت داده مي‌شود. pa را اشاره‌گر a مي‌نامند، زيرا به محلي از حافظه اشاره مي‌كند كه مقدار متغير a در آن ذخيره شده است. به هرحال به‌خاطر بسپاريد كه pa مقدار a را معرفي نمي‌كند، بلكه آدرس a را معرفي مي‌كند و به همين لحاظ آن را متغير اشاره‌گر نامند. شكل 1.8 رابطة بين pa و a را نشان مي‌دهد.



شكل 1.8 رابطة بين اشاره گر و متغير

داده‌اي كه با a معرفي مي‌گردد (يعني داده‌اي كه در خانة a از حافظه ذخيره شده است) با عبارت *pa در دسترس قرار مي‌گيرد كه در آن * اپراتور يكاني يا تك اپراندي است كه فقط روي متغيرهايي از نوع اشاره گر عمل مي كند. بنابراين a و *pa هر دو همان قلم داده (يعني هر دو محتواي خانه‌هاي يكسان از حافظه) را معرفي مي‌كنند. پس با اجراي دو دستور

pa = &a ;
k = *pa ;

k و a هر دو يك مقدار را معرفي خواهند كرد ؛ يعني مقدار a به طور غيرمستقيم به k نسبت داده خواهد شد . به عبارت ديگر، نتيجة دو دستور مزبور مشابه نتيجة دستور k = a است.

بنابراين عملگر * در مورد *pa محتواي محلي را برمي گرداند كه آدرس آن در pa قرار دارد و به همين لحاظ به آن عملگر غيرمستقيم نيز گويند.

آدرس داده‌ها

هر متغيري داراي آدرس منحصر به فردي است كه محل آن متغير را در حافظه مشخص مي‌كند. در بعضي كاربردها بهتر است كه براي دستيابي به متغير به جاي نام آن متغير، از آدرس آن استفاده كرد. براي به دست آوردن آدرس متغير، اپراتور ampersand يا & به كار مي‌رود. براي مثال، فرض كنيد كه متغير k از نوع long int و آدرس آن 1004 باشد، دستور Ptr = &A; مقدار 1004 (آدرس متغير A) را در متغير Ptr ذخير مي‌كند كه البته بايد Ptr از نوع اشاره‌گر توصيف شده باشد كه به متغيري از نوع long int اشاره مي‌كند.

مثال 1.8 برنامة سادة زير مقدار و آدرس متغير A را چاپ مي‌كند.

```
# include <stdio.h>
main ()
  {
    int A = 5 ;
    printf (" The value of A is:  %d\n" , A) ;
    printf (" The address of A is:  %p\n" , &A) ;
  }
```

خروجي برنامه

```
The value of A is:  5
The address of A is:  1004
```

يادآور مي‌شويم كه در تابع printf براي چاپ آدرس متغير از كد فرمت %p استفاده شده است. اين كد فرمت ممكن است در كامپايلرهاي قديمي وجود نداشته باشد.

همچنين مي‌توان قطعه برنامة بالا را به صورت زير نيز نوشت.

```
# include <stdio.h>
main ()
  {
    int A = 5 ;
    int *pA ;
    pk = &A ;
    printf ("the address of A is:  %p\n" , pA) ;
  }
```

خروجي اين برنامه نيز مشابه قبلي خواهد بود.

از مطالب مطرح شده مي‌توان نتيجه گرفت كه عملگر ستاره يعني * به دو مفهوم جداگانه به كار مي‌رود.

الف) در معرفي متغيرهاي عملگر، اشاره‌گر در سمت چپ متغيرهاي مورد نظر قرار مي‌گيرد، مانند مثالهاي زير.


```
int *p1 , *p2 , *p3 ;
```

```
float *p4 , *p5 ;
char *p6 , *p7 ;
```

ب) براي دستيابي به مقدار متغيري كه آدرس آن در متغير اشاره‌گر قرار دارد، مانند

```
p1 = &a ;
*p1 = a ;
```

ϖ

ϖ **مثال 2.8** به برنامهٔ زير توجه كنيد.

```
# include <stdio.h>
main ()
  {
    char * pch ;
    char ch1 = `Z´ , ch2 ;
    printf ("the address of pch is %p" , &pch) ;
    pch = &ch1 ;
    printf ("the value stored at pch is %p\n" , pch) ;
    printf ("the value stored at the address pointed by pch is %c\n" , *pch) ;
    ch2 = *pch ;
    printf ("the value stored at ch2 is %c\n" , ch2) ;
  }
```

خروجي برنامه

```
the address of pch is 1004
the value stored at pch is 2001
the value stored at the address pointed by pch is Z
the value stored at ch2 is Z
```

در اين برنامه متغير pch اشاره‌گري به متغيرهايي از نوع كاراكتر توصيف شده است. متغيرهاي ch1 و ch2 نيز از نوع كاراكتر اعلان شده‌اند كه به متغير ch1 مقدار اولية كاراكتر `a´ نسبت داده شده است. در دستور printf اول آدرس متغير pch چاپ مي‌گردد كه به فرض 1004 است. سپس آدرس متغير ch1 به pch نسبت داده مي شود. در دستور printf دوم، مقدار pch (آدرس متغير ch1) كه به فرض 2001 است چاپ مي‌گردد. در دستور printf سوم، محتواي خانه‌اي از حافظه كه آدرس آن در متغير pch قرار دارد (يعني مقدار متغير ch1) و كاراكتر `a´ است چاپ مي‌شود. سپس در خط بعدي همين مقدار (يعني حرف `a´) به متغير ch2 نسبت داده مي‌شود. بالاخره با دستور printf آخري مقدار متغير ch2 (كه همان حرف `a´ است) چاپ مي‌گردد.

نكته. عبارت *pch در سمت چپ دستور جايگذاري نيز ظاهر می‌شود. مثلاً در همان برنامهٔ بالا پس از اجراي دستور pch = & ch1 ; با دستور *pch = `b´ ; كاراكتر b به متغير ch1 نسبت داده مي‌شود.

ϖ

مقداردهي اوليه به اشاره‌گر

به هر نوع متغير از نوع اشاره گر مي‌توان هنگام اعلان آنها، مشابه ساير متغيرها ، مقدار اوليه نيز نسبت داد . در اين صورت مقدار اوليه مورد نظر بايد يك آدرس باشد. پس اشاره‌گر NULL يا يك آدرس را به عنوان مقدار اوليه می‌پذيرد. براي مثال مي‌توان دستورهايي به صورت زير نوشت.

```
int x ;
int *px = &x ;
```

اما نمي‌توان متغيری را قبل از اينكه توصيف يا اعلان گردد در دستوري به كار برد. بنابراين مجموعه دستورهاي زير قابل قبول نيست.

```
int *px = &x ;
int x ;
```

همچنين مي‌توان اشاره‌گر را به صورت int *ptr = 0 ; مقداردهي اوليه كرد كه براي مشخص ساختن بعضي شرايط خاص ب ه كار برده مي‌شود.

در حالت كلي، نسبت دادن مقدار صحيح به متغير اشاره‌گر مفهوم ندارد. به هرحال، مثال اخير حالت استثنايی در اين مورد است كه همان طور كه گفتيم، براي مشخص ساختن بعضي شرايط خاص به كار می‌رود. در چنين مواردي توصيه مي‌گردد كه ثابت سمبوليكی مانند NULL را كه معرف صفر باشد تعريف كرد و آن را به اشاره‌گر اختصاص داد. اين روش تأكيد مي‌كند كه اختصاص دادن صفر، معرف شرطی ويژه است.

ϖ **مثال 3.8** برنامه‌ای به زبان C ممكن است تعاريف و عبارات زير را شامل باشد.

```
# define NULL 0
float x , y ;
float *pr = NULL ;
```

در اين مثال متغيرهاي x و y به صورت متغيرهايي از نوع مميز شناور و pr به صورت متغير اشاره‌گر اعلان شده كه مقداري ويژه به عنوان مقدار اوليه به آن نسبت داده شده است . بنابراين استفاده از ثابت سمبوليك NULL نشان مي‌دهد كه اين اختصاص مقدار اوليه، چيزي به غير از اختصاص مقدار صحيح معمولي است . به هرحال در اغلب كامپايلرهاي C ثابت سمبوليك NULL در چندين header file و بويژه در <stdio.h> تعريف شده است. پس اختصاص مقدار اولية صفر يا NULL به يك اشاره‌گر هم‌ارز است، ولي NULL ترجيح داده مي‌شود.

ϖ

اشاره‌گر تهي

زبان C مفهوم اشاره‌گر NULL را پشتيباني مي‌كند و آن اشاره‌گري است كه به هيچ شئي قابل‌قبول يا معتبر اشاره نمي‌كند. اشاره گر NULL هر اشاره‌گري است كه مقدار صحيح صفر به آن نسبت داده شده باشد.

ϖ **مثال 4.8** در مثال زير اشاره‌گر p مقدار صفر دارد.

```
char *p ;
p = 0 ;
```

ϖ

اشاره‌گر NULL بويژه در دستورهاي مربوط به كنترل جريان مفيد است، زيرا اشاره گرهايي با مقدار صفر false درنظر گرفته مي شوند، درحالي كه متغيرهاي اشاره‌گر با ساير مقادير true منظور مي‌گردند.

ϖ **مثال 5.8** در برنامهٔ زير حلقهٔ while تا موقعي كه p اشاره‌گر NULL نباشد، عمل تكرار را ادامه مي دهد.

```
char *p ;
 ....
 ....
while (p)
 {
 ....
 ....
 }
```

اين گونه كاربرد اشاره گرها، بويژه در كاربردهايي آشكار مي‌گردد كه آرايه‌هايي از اشاره گرها را به كار مي‌برد، و در همين فصل بررسي می‌كنيم.

ϖ

عمليات روي اشاره‌گرها

عمليات متداولي كه روي اشاره‌گرها انجام مي‌شوند عبارت‌اند از انتساب، محاسبه، و مقايسه كه به شرح هر يک می‌پردازيم.

الف) انتساب

در مورد اشاره‌گرها نيز مشابه ساير انواع متغيرها، مي توان مقداري را به متغير اشاره‌گر نسبت داد. اما به طوري كه گفتيم، اين عمل مانند ساير متغيرها گسترده نيست. به متغير اشاره‌گر مي‌توان آدرس متغير يا مقدار صفر نسبت داد.

ب) محاسبه

زبان C اجازه مي‌دهد كه مقدار صحيح‌ی را به اشاره‌گر اضافه و يا از آن كسر كنيد. براي مثال اگر p متغير اشاره‌گر باشد، عباراتي مشابه p+5 و p-5 معتبر است. ولي بايد به مفهوم آن دقت كافي شود . براي مثال مرفهوم p+5 آن است كه به پنج شئي بعد از شئي كه p به آن اشاره مي‌كند اشاره خواهد كرد. بنابراين اگر p آدرس متغيري از نوع short int p ; را داشته باشد كه دو بايت حافظه اشغال مي‌كند، عبارت p+5 به (10 = 2×5) 10 بايت بعد از آدرسي كه p معرف آن است اشاره خواهد كرد. حال اگر p آدرس متغيري از نوع float p ; را در خود داشته باشد، عبارت p+5 به (20 = 4×5) 20 بايت بعد اشاره خواهد كرد. بنابراين p+5 هميشه به مفهوم به اندازهٔ 5 شئي بعد از آنكه p اشاره مي‌كند خواهد بود. پس كامپايلر 5 را در بزرگي يا طول شئي مورد نظر برحسب بايت ضرب مي‌كند و آن را بر محتواي p اضافه مي‌كند تا آدرس جديد به دست آيد. بنابراين عمليات محاسباتي روي اشاره‌گرها، چهار عمل افزودن، كاستن، ++ و -- است.

اگر دو متغير اشاره گر از يك نوع باشند، يعني اشيايي كه اشاره گرهاي مزبور به آن اشاره مي كنند يكسان باشند، مي توان مقادير آن دو اشاره‌گر را از هم تفريق كرد . براي مثال مقدار &a[0] - &a[3] برابر 3 خواهد بود. در واقع اين تفاضل تعداد اشياء بين اين دو اشاره گر را معرفي مي‌كند.

ϖ **مثال 6.8** به برنامهٔ زير توجه كنيد.

```
#include<stdio.h>
main ()
  {
     int *px , *py ;
     static int A[6] = {1 , 2 , 3 ,4 , 5 , 6};
     px = &A[0] ;
     py = &A[5] ;
     printf("px=%x py=%x" , px , py) ;
     printf("\n py – px =%x" , py - px) ;
  }
```

خروجي خط اول برنامه آدرس دو متغير px و py بر حسب هگزادسيمال خواهد بود و خروجي خط دوم مقدار 5 است.

ϖ

ج) مقايسه

متغيرهاي اشاره‌گر را، كه هر دو به داده هايي از يک نوع مشابه اشاره داشته باشند، مي توان با هم مقايسه كرد. به عبارت ديگر دو اشاره‌گر را مي‌توان در يك عبارت رابطه‌اي با يكديگر مقايسه كرد.

ϖ **مثال 7.8** فرض كنيد px و py اشاره‌گرهايي باشند كه به عناصري از يک آرايه اشاره مي كنند. چند عبارت منطقي حاصل از اين دو متغير به اين شكل خواهد بود.

```
px < py
px >= py
px = = py
```



```
px != py
```

```
px = = null
```

همچنین دستورهای زیر درست است.

```
if (px<py)
   printf ("px points to lower memory than py") ;
else
   printf ("px points to upper memory than py") ;
ϖ
```

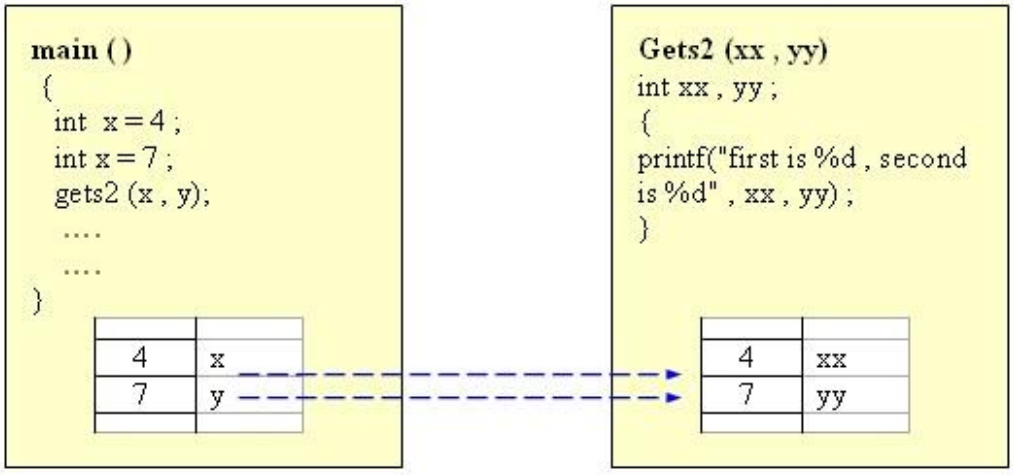
انتقال مقادیر به تابع

حال ببینیم وقتي مقاديري را به تابع گذر مي دهيم چه اتفاقي رخ مي‌دهد. در اينجا برنامة ساده‌اي را ملاحظه مي‌كنيد كه دو مقدار صحيح 4 و 7 را به تابعي به نام gets2 مي‌فرستد.

```
main ()
   {
      int x = 4 , y = 7 ;
      gets2 (x , y) ;
   }
void gets2(xx , yy)        /* print out values of two arguments */
int xx , yy ;
   {
      printf ("first is %d , second is %d" , xx , yy) ;
   }
```

اين تابع عمل خاصي انجام نمي دهد، فقط دو مقداري را كه به آن گذر داده شده است چاپ مي كند. اما تابع مزبور نكتة مهمي را نشان مي‌دهد و آن اينكه تابع دو مقدار از برنامة فراخوانندة آن دريافت مي كند و آنها را به طور جداگانه، يعني به صورت دوبله، در فضاي حافظة خاص خودش ذخيره مي كند. حتي تابع مي تواند به آن دو مقدار، اسامي متفاوتي (مانند مثال مورد نظر ما) كه فقط در تابع مزبور شناخته شده است اختصاص دهد كه در اينجا اين اسامي جديد نيز xx و yy است و به جاي x و y به كار رفته است (البته مي‌توانست همان x و y نيز به كار برده شود).

شكل 2.8 اين سازوكار را نمايش مي‌دهد. حال اين تابع مي‌تواند روي متغيرهاي جديد xx و yy، بدون اينكه تأثيري روي x و y داشته باشد، هر عملي را انجام دهد (اگر اسامي يكسان انتخاب مي‌شد، باز هم در شيوة كار تغييري حاصل نمي‌شد).



شكل 2.8 انتقال مقادير به تابع

انتقال اشاره‌گر به تابع

اشاره‌گرها اغلب به عنوان آرگومان به يك تابع فرستاده مي شود. اين امر اجازه مي دهد كه عناصر داده هاي برنامة فراخواننده، كه معمولاً تابع main است، با تابع فراخوانده شده قابل دستيابي باشند و در داخل تابع فراخوانده شده تغيير يابند و نتيجه در تابع يا برنامة فراخواننده نيز اعمال گردد. اين گونه كاربرد اشاره‌گر، گذر دادن آرگومانها با آدرس و يا گذر دادن آرگومانها با مرجع ناميده مي‌شود.

هنگامي كه آرگومانها با مقدار گذر داده مي شوند، عناصر داده (مقدار داده) به تابع كپي مي شوند. بنابراين هر گونه تغييرات اعمال شده در روي آنها در درون تابع يا روتين فراخواننده اثر نمي گذارد. اما وقتي كه آرگومان به صورت آدرس انتقال مي يابد (يعني وقتي كه اشاره گر به تابع گذر داده مي‌شود)، در واقع آدرس آن قلم از داده به تابع فرستاده مي شود. حال محتواي آن آدرس به راحتي هم در درون تابع مزبور و هم در تابع فراخواننده قابل دستيابي است . به‌علاوه هر تغييراتي كه روي آن قلم داده انجام پذيرد (يعني هر گونه تغييراتي كه در محتواي آدرس مورد نظر انجام گيرد) هم در تابع فراخوانده شده و هم در برنامه يا تابع فراخواننده تأثير مي گذارد و تشخيص داده مي‌شود.

دو برنامة نمايش داده شده در مثالهاي 8.8 و 8.9 دو گونه از تابعي به نام add را نشان مي دهد كه آرگومان خود را يك واح د افزايش مي‌دهد. مثال 8.8 متغير count را با روش فراخواني با مقدار، به تابع گذر مي دهد. تابع add آرگومان خود را يك واحد افزايش مي دهد و مقدار جديد را با دستور return به تابع main برمي‌گرداند. مقدار جديد در تابع main به count نسبت داده مي‌شود. در مثال 8.9، برنامة مورد نظر متغير count را با فراخواني آدرس گذر مي دهد؛ يعني آدرس count (نه مقدار آن) به تابع add گذر داده مي‌شود. تابع add1 اشاره‌گری به مقدار صحيح را كه در اينجا countptr ناميده شده است به عنوان آرگومان دريافت مي كند. تابع add مقداري را كه با countptr به آن اشاره شده است (يعني محتواي محلي را كه آدرس آن در اشاره گر countptr قرار دارد) يك واحد افزايش مي دهد. اين عمل همچنين مقدار count را در main تغيير مي‌دهد.

ϖ **مثال 8.8** مقدار متغير با به کار بردن فراخواني با مقدار افزايش می‌يابد.

```
#include <stdio.h>
main ()
   {
      int count = 7 ;
      int add(int) ;
      printf ("the original value of count is %d\n" , count) ;
      count = add (count) ;
      printf ("the new value of count is %d\n" , count) ;
   }
int add(int c)
   {
      return ++c ; \* incrementsl variable c */
   }
```

خروجی برنامه

```
the original value of count is 7
the new value  of count is 8
```

ϖ

ϖ **مثال 9.8** مقدار متغير با فراخواني آدرس افزايش می‌يابد.

```
# include <stdio.h>
main ()
   {
      int count = 7 ;
      int add (int *) ;
      printf("the original value of count is %d\n" , count) ;
      add (& count) ;
      printf ("the new value of count is %d/n" , count) ;
   }
void add (int *countptr)
   {
      + + (*countptr) ; /* increments count in main */
   }
```

خروجی برنامه

```
the original value of count is 7
the new value of count is 8
```

پارامتر متناظر تابعي که آدرس‌ی را به عنوان آرگومان دريافت مي کند، بايد اشاره گر باشد. مثلاً عنوان تابع add در اين مثال به صورت زير است.

```
void add (int *countptr)
```

و بيان مي‌کند که add آدرس متغيری از نوع int را دريافت و در countptr ذخيره خواهد کرد.

ϖ

ϖ **مثال 10.8** برنامة زير تفاوت بين آرگومانهاي مرعمولي را که با مقدارعبور داده مي‌شوند و آرگومانهاي اشاره گر را که با مرجع عبور داده مي‌شوند روشن مي‌سازد.

```
#include<stdio.h>
main ()
  {
      int u = 1 ;
      int v = 3 ;
      void func1 (int u , int v) ;
      void func2 (int *pu , int *pv) ;
      printf (" Before calling func1:  u=%d v=%d " , u , v) ;
      func1(u , v) ;
      printf (" After calling func1:  u=%d v=%d " , u , v) ;
      printf (" Before calling func2:  u=%d v=%d " , u , v) ;
      func2(u , v) ;
      printf (" After calling func2:  u=%d v=%d " , u , v) ;
  }
void func1 (int u , int v)
  {
      u = 0 ;
      v = 0 ;
      printf (" Within func1:  u=%d v=%d " , u , v) ;
      return ;
  }
```

```
void func2 (int *pu , int *pv)
  {
     *pu = 0 ;
     *pv = 0 ;
     printf (" Within func2:  *pu=%d *pv=%d " , *pu , *pv) ;
     return ;
  }
```

خروجي برنامه به اين شكل خواهد بود.

```
Before calling func1:  u = 1 v = 3
Within func1:  u = 0 v = 0
After calling func1:  u = 1 v = 3
Before calling func2:  u = 1 v = 3
Within func2:  *pu = 0 *pv = 0
After calling func2:  u = 0 v = 0
```

ملاحظه مي‌كنيد كه تابع func1 دو متغير از نوع صحيح را به عنوان آرگومان مي پذيرد. تابع func2 دو اشاره‌گر به متغيرهاي صحيح را به عنوان آرگومان دريافت مي‌دارد.

ϖ

انتقال دوطرفة اطلاعات

وقتي كه تابعي آدرس متغيري را در برنامة فراخوانندة آن بداند، مي‌تواند هم مقاديري را در اين متغيره ا قرار دهد (يعني مقادير آنها را تغيير دهد) و هم مقادير آن متغيرها را به كار برد. بنابراين به كمك اشاره گرها مي‌توان مقادير را هم از برنامة فراخواننده به تابع فراخوانده شده و هم از تابع فراخوانده شده به برنامة فراخوانندة آن (درواقع در هر دو جهت) گذر داد. البته در مبحث توابع، انتقال مقادير به روش فراخواني با مقدار بررسي شد، ولي روش مذكور ما را قادر مي سازد كه بيش از اين مقدار را به برنامه يا تابع فراخواننده برگردانيم و بدين طريق محدوديتي كه در برگرداندن نتايج تابع فرعي به كمك نام آن تابع وجود دارد و در آن فقط مي‌توان يك مقدار را با نام تابع برگرداند از بين برد.

ϖ **مثال 11.8** برنامة زير ضرايب معادلة درجه دومي را مي‌خواند و سپس با فراخوانده شدن تابع فرعي اى به نام root ريشه‌هاي معادلة مزبور را محاسبه مي‌كند و به تابع اصلي بر مي‌گرداند. اگر معادله ريشة حقيقي نداشته باشد، تابع فرعي هر دو ريشه را صفر بر مي‌گرداند. در ضمن اگر معادله ريشه داشته باشد، ضرايب معادله همراه با ريشه هاي آن در تابع اصلي چاپ مي‌شود. در غير اين صورت، ضرايب آن همراه با پيغام مناسب چاپ مي‌شود.

```
#include <stdio.h>
# include <math.h>
main ()
   {
     float a , b , c , x1 , x2 ;
     scanf ("%f %f %f" , &a , &b , &c) ;
     root (a , b , &x1 , &x2) ;
     if (x1 = = 0 && x2 = = 0)
        printf ("\n %f %f %f no real solution" , a , b , c) ;
     else
        printf ("\n %f %f %f %f %f" , a , b , cx1 , x2) ;
   }
void root (a , b , c , px1 , px2)
float a , b , c , *px1 , *px2 ;
  {
     float d , delta ;
     delta = b*b - 4*a*c ;
     if (delta < 0)
        { *px1 = *px2 = 0 ;
          return ;
        }
     else
        { d = sqrt (delta) ;
          *px1 = (-b+d) / (2*a) ;
          *px2 = (-b-d) / (2*a) ;
          return ;
        }
   }
```

در فراخواني تابع root آدرس متغيرهاي x2 و x1 (كه بايد ريشه‌هاي معادله را بپذيرد) به تابع مذكور گذر داده مي شود و سپس در تابع root، اگر معادله داراي ريشه هاي حقيقي باشد، مقادير آنها به متغيرهاي x2 و x1 نسبت داده مي‌شود (يعني در محلهايي كه آدرس آنها در متغيراشاره‌گر px1 و px2 است قرار مي‌گيرد). در غير اين صورت به هر دو متغير مقدار صفر نسبت داده مي شود. بدين طريق بيش از يك مقدار از تابع فرعي به اصلي برگردانده مي‌شود.

ϖ

اشاره‌گرها و آرايه‌ها

بين اشاره‌گرها و آرايه‌ها رابطة نزديكي وجود دارد. نام آرايه در واقع اشاره‌گري به اولين عنصر آن است. قطعه برنامة زير را درنظر بگيريد.

```
char str[80] , *p ;
p = str ;
```

مي‌دانيم كه نام آرايه همان آدرس اولين عنصر آرايه است. بنابراين دو دستور

```
p = &str[0] ;
p = str ;
```

هم‌ارزند. پس در قطعه برنامهٔ بالا در اشاره‌گر p، آدرس آرايه (يعني آدرس اولين عنصر آرايه) قرار داده شده است . حال اگر بخواهيم به پنجمين عنصر در str دسترسي داشته باشيم، مي‌توانيم اين كار را به دو روش زير انجام دهيم.

```
str[4]
```

يا

```
 *(p+4)
```

هر دو دستور، پنجمين عنصر را برمي‌گردانند.
همين طور اگر داشته باشيم

```
int a[15] , *p ;
p = &a[0] ;
```

دو عبارت a[3] و *(p+3) هم‌ارزند و هر دو چهارمين عنصر از 15 عنصر را برمي‌گردانند؛ يعني، به طور كلي عنصر a[k] هم‌ارز با *(p + k) خواهد بود و هر دو محتواي خانه k اُم آرايهٔ a يا (k+1) اُمين عنصر از مجموعه عناصر مزبور را برمي گردانند. همين طور عبارت مزبور هم ارز *(a + k) است، زيرا a نام آرايه، معرف آدرس آغاز آن است و (a + k) آدرس خانهٔ k اُم آرايه خواهد بود و در نتيجه عنصر *(a + k) محتواي خانهٔ k اُم از آرايهٔ مزبور را برمي‌گرداند. بنابراين C، سه روش براي دستيابي به عناصر آرايه در اختيار ما قرار مي‌دهد. اما مهم است بدانيم كه دستيابي به عناصر آرايه از طريق اشاره‌گر، سريع‌تر از روش استفاده از انديس است . لذا روش *(p + k) و همين طور *(a + k) سريع‌تر از a[k] عمل مي‌كند. بدين لحاظ استفاده از اشاره‌گرها براي دستيابي به عناصر آرايه، روش بسيار متداولي در زبان C است.

حال براي تفسير k در عبارتهاي *(p+k) و *(p+k) به دو برنامهٔ زير توجه كنيد.

```
main ()
   {
      static int nums[ ] = {92 , 81 , 70 , 69 , 58} ;
      int k ;
      for (k = 0 ; k<5 ; k+ +)
      printf (" %d\n " , nums[k]) ;
   }
```

خروجي برنامه

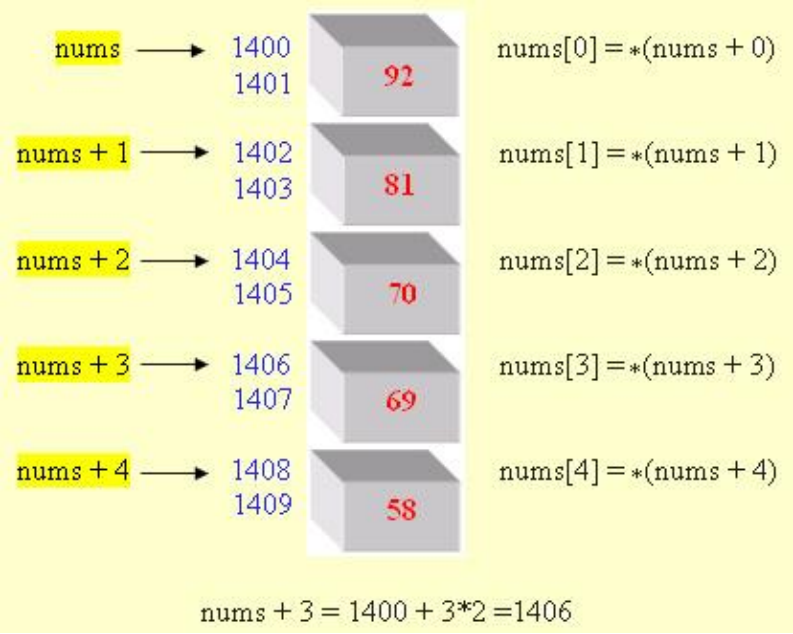
```
92
81
70
69
58
```

برنامهٔ مزبور برنامهٔ ساده‌ای است كه در آن براي دستيابي به عناصر آرايه از روش متعارف علامتگذاري آرايه استفاده شده است .

حال همان برنامه با روش به كارگيري از اشاره‌گر به صورت زير خواهد بود.

```
/* uses pointers to print out values from array */
main ()
   {
      static int nums[ ]= {92 , 81 , 70 , 69 , 58}
      int k ;
      for (k=0 ; k<5 ; k++)
      printf (" %d\n ", *(nums + k)) ;
   }
```

شكل 3.8 نشان مي‌دهد كه منظور از *(nums+k) محتواي k خانه، بعد از nums است كه در آن بزرگي هر خانه مساوي بزرگي داده يا شئي مورد نظر در آرايه برحسب بايت است كه در مثال بالا 2 بايت است. همچنين در عبارت *(nums + k) نقش اپراتور ستاره را به‌عنوان عملگر غيرمستقيم ملاحظه مي‌كنيد كه محتواي خانه يا آدرس nums+k را در اختيار قرار مي‌دهد.



شکل 3.8 جمع اشاره‌گرها (آدرسها و مقادير)

از موارد بالا نتيجه مي‌شود كه array[index] با *(array + index) يكسان است.
همچنين دو راه براي مراجعه به آدرس عنصر ی از آرايه وجود دارد. يكي به صورت nums+k در فرم اشاره‌گر و ديگري به صورت nums[k] در فرم آرايه.
حال اجازه دهيد با برنامه‌ای ساده، رابطة بين عناصر آرايه و آدرس آنها را بررسي كنيم.
ϖ **مثال 12.8** برنامة زير را درنظر بگيريد.

```
# include <stdio.h>
main ()
  {
    static int x[6] = {10 , 11 , 12 , 13 , 14 , 15};
    int i ;
    for (i=0 ; i<6 ; + +i)
      printf("\n i =%d x[i] = %d *(x+i) = %d &x[i] = %x x+i =%x", i , x[i] , *(x+i) , &x[i] , x+i) ;
  }
```

(فرض بر اين است كه آدرس شروع آرايه، 72 در مبناي 16 است.)

خروجي برنامه

i = 0	x[i] = 10	*(x+i) = 10	&x[i] = 72	x+i = 72
i = 1	x[i] = 11	*(x+i) = 11	&x[i] = 74	x+i = 74
i = 2	x[i] = 12	*(x+i) = 12	&x[i] = 76	x+i = 76
i = 3	x[i] = 13	*(x+i) = 13	&x[i] = 78	x+i = 78
i = 4	x[i] = 14	*(x+i) = 14	&x[i] = 7a	x+i = 7a
i = 5	x[i] = 15	*(x+i) = 15	&x[i] = 7c	x+i = 7c

از خروجي بالا فرق بين x[i] كه معرف i اُمين عنصر آرايه است، با &x[i] كه آدرس آن را نمايش مي دهد، مشخص مي گردد. در ضمن مشاهده مي‌شود كه مقدار i اُمين عنصر آرايه را مي‌توان با x[i] يا *(x + i) معرفي كرد. در ضمن مي‌توان تفاوت بين x + i و *(x + i) را ملاحظه كرد كه اولي معرف آدرس و دومي نشان دهندة محتواي آن آدرس است . همچنين نتيجه گرفته مي شود كه اگر x[i] در سمت چپ يك دستور جايگذاري باشد، مي توان به جاي آ ن *(x + i) را به كار برد. اصولاً همه جا مي توانيم به جاي x[i] همارز آن *(x+i) را به كار ببريم. به هرحال عباراتي مشابه x و x + i و &x[i] كه معرف آدرس اند، نمي‌توانند در سمت چپ دستور جايگذاري به كار روند. همچنين آدرس آرايه نمي تواند به طور دلخواه تغيير يابد. بنابراين عبارتي مشابه x++ مجاز نيست.
قبلاً گفتيم كه نام آرايه، به طور واقعي اشاره‌گری به اولين عنصر آرايه است. در نتيجه بايد امكان داشته باشد كه يك آرايه را، به جاي روش قراردادي متداول، متغير اشاره‌گر تعريف كرد.
به هرحال تعريف آرايه به روش قراردادي موج ب مي گردد كه بلوك ثابت از حافظه، در آغاز اجراي برنامه رزرو شود . ولي اگر آرايه برحسب متغير اشاره‌گر توصيف شود، با اين عمل رزرو كردن جا اتفاق نمي افتد. درنتيجه موقع استفاده از اشاره گر براي معرفي آرايه، نياز است كه به طريقي قبل از اينكه عناصر آرايه مورد پردازش قرار گيرد، به عناصر آرايه، حافظه اختصاص داده شود . در حالت كلي اختصاص اولية حافظه در چنين مواردي، با استفاده از تابع كتابخانه‌اي malloc انجام مي‌گيرد. گرچه شيوة انجام اين كار از كاربردي به كاربرد ديگر فرق خواهد كرد .
اگر آرايه به صورت متغير اشاره گر تعريف گردد، نمي توان به عناصر آرايه هاي عددي مقدار اوليه نسبت داد . در نتيجه اين گونه موارد، به تعريف آرايه به صورت روش عادي و قراردادي نياز دارد.
ϖ **مثال 13.8** اگر بخواهيم a را به صورت آرايه 10 عنصري از مقادير صحيح تعريف كنيم، مي‌توان آن را به جاي int a[10] ; به صورت int *a ; نوشت؛ يعني a را متغير اشاره‌گر تعريف كرد.

به هرحال در روش دوم، به a كه آرايه‌ای 10 عنصري است يك بلوك حافظه اختصاص داده نمي‌شود، بلكه a به صورت متغير اشاره‌گر تعريف

شده است.
براي اختصاص حافظة مورد نياز به a جهت معرفي آرايه‌اى 10 عنصري، مي‌توان از تابع malloc به صورت زير استفاده كرد.

```
a = malloc (10 * sizeof(int)) ;
```

اين تابع بلوك حافظه اى براي ذخيره كردن 10 مقادير صحيح رزرو مي كند. در دستور بالا اپراتور sizeof بزرگي نوع داده int را برحسب بايت برمي‌گرداند. اين مقدار در 10 (تعداد عناصر آرايه) ضرب مي‌شود تا فضاي مورد نياز برحسب بايت تعيين و رزرو شود . a به صورت اشاره‌گر به مقدار صحيح تعريف شده است و تابع malloc يك اشاره‌گر به كاراكتر برمي‌گرداند و مي‌دانيم كه در زبان C مقادير صحيح و كاراكترها معادل‌اند. لذا دستور بالا قابل قبول است. اگرچه از لحاظ اطمينان كامل مي‌توان از تبديل نوع cast استفاده كرد و آن را به صورت زير به كار برد.

```
a = (int *) malloc (10 * sizeof (int)) ;
```

اين گونه اختصاص حافظه به روش اختصاص حافظه به صورت پويا موسوم است.
به هرحال اگر قرار باشد به عناصر آرايه، مقدار اوليه نيز اختصاص يابد، بايد a به جاي متغير اشاره گر به صورت آرايه توصيف گردد ، مشابه زير.

```
int a[10] = {1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10} ;
```

يا

```
int a[ ] = {1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10}
```

ϖ
در برنامه‌نويسي با C ممكن است براي مراجعه به عناصر آرايه ، به جاي روش معمول، عباراتي برحسب اشاره‌گرها به كار ببريم. اين روش در آغاز كار كمي غيرطبيعي به نظر مي‌آيد، ولي مي‌توان با كمي تمرين به سادگي تجربة لازم را در اين مورد به دست آورد. در مثال 14.8 ضمن اينكه روش استفاده از تابع malloc نشان داده مي‌شود، عناصر آرايه نيز با استفاده از اين روش در دسترس قرار می‌گيرند.
ϖ **مثال 14.8** برنامة زير مجموعه‌اى از اعداد را از ورودي می‌خواند و با استفاده از اشاره‌گرها مرتب مي‌كند.

```
# include <stdio.h>
main ()
  {
    int k , m , *a ;
    void sort (int k , int *a) ;
    scanf ("%d" , &m) ;    /* read in a value for m */
    a = (int*) malloc (m * sizeof(int)) ;        /* allocate memory */
    for (k = 0 ; k<m ; + +k)        /* read in the list of numbers */
      scanf ("%d" , a + k) ;
    sort (m , a) ;
    for (k=0 ; k<m ; + +k)          /* display sorted list , elements */
      printf ("\n k=%d a =%d" , k+1 , *(a+k)) ;
  }
void sort (int m , int *a)          /* sort array in ascending order */
  {
    int i , j , temp ;
    for (i=1 ; i<m ; + +i)
      for (j=0 ; j<m-i ; + +j)
        if (*(a+j) < *(a+j+1))
          {
            temp = *(a+j) ;
            *(a+j) = *(a+j+1) ;
            *(a+j+1) = temp;
          }
    return ;
  }
```

در اين برنامه، آرايه اي با عناصري از نوع مقادير صحيح، ب ه صورت اشاره‌گر به مقدار ى صحيح تعريف شده است . در آغاز به كمك تابع كتابخانه‌اي malloc به متغير اشاره‌گر، حافظه اختصاص داده شده است (يعني حافظة مورد نياز از سيستم گرفته شده و آدرس اولين بايت آن در a قرار داده شده است). در هر دو تابع اصلي و فرعي براي پردازش هر عنصر از روش مراجعه به اشاره گر استفاده شده است. ملاحظه مي‌كنيم كه در تابع scanf نيز براي آدرس عنصر k اُم به جاي &a[k] از a + k استفاده شده است. به طريق مشابه، در تابع printf براي معرفي مقدار k اُمين عنصر، به جاي a[k] از *(a + k) استفاده شده است. ملاحظه مي‌كنيد كه در تابع فرعي نيز آرگومان آن ، به جاي آرايه، متغير اشاره تعريف شده است.

ϖ

اشاره‌گرها و آرايه‌هاي چند بعدي

ديديم كه عناصر آرايه‌اى يك بعدي مي‌تواند برحسب اشاره‌گر (نام آرايه) و مقدار به عنوان آفست به منظور جبران كردن مقدار انديس عنصر مورد نظر آرايه نمايش داده شود . مثلاً اگر نام آرايه a و عنصر مورد نظر ما a[5] باشد، مي‌توان به آن به صورت *(a + 5) مراجعه كرد كه در آن مقدار آفست همان 5 است كه به نام آرايه افزوده شده است و به كمك عملگر * به مقدار آن دسترسي پيدا مي‌كنيم.
حال مي‌توان گفت كه آرايه اى دوبعدي نيز مجموعه‌اي از آرايه‌هاي يك‌بعدي است. بنابراين مي‌توان آرايه‌اى دو بعدي را به صورت اشاره‌گر به گروه پيوسته و مجاور هم از آرايه هاي يك بعدي تعريف كرد . درنتيجه مي توان توصيف آرايه اى دو بعدي را ب ه جاي data-type array[d1][d2] ; (كه در آن منظور از d1 و d2 به‌ترتيب بزرگي ابعاد اول و دوم آرايه است) به صورت زير نوشت.

```
data-type (*ptvar)[d2] ;
```

اين ايده را مي‌توان به آرايه‌هاي n بعدي تعميم داد و آن را به جاي

```
data-type array [d1][d2]...[dn] ;
```


به صورت زير نوشت.

data-type (*ptvar)[d2][d3]...[dn] ;

که در آنها data-type نوع عناصر آرایه و array نیز نام آرایه است . عناصر d1 , d2 ,..., dn نیز به ترتیب ماکزیمم عناصر هر اندیس یا هر بعد آرایه‌اند. درضمن توجه کنید که ptvar نیز نام متغیر اشاره‌گر است.

انتقال آرایه به تابع

در زبان C، نام هر آرایه‌اي که به عنوان آرگومان تابع به کار رود، آدرس اولین عنصر آرایه تفسیر مي‌گردد. برنامهٔ زیر را درنظر بگیرید.

```
main ()
  {
    float func() ;
    float x , array[15] ;
    .....
    .....
    x = func(array) ; /* same as func (&array [0]) */
    .....
    .....
  }
```

حال در تابع فرعي نیاز است که آرگومان را ب ه عنوان اشاره‌گر به اولین عنصر آرایه توصیف کنیم . براي این کار، دو راه به صورت زیر وجود دارد.

راه اول

```
func(ar)
float ar[ ] ;
   {
     .....
     .....
   }
```

راه دوم

```
func(ar)
float *ar ;
   {
     .....
     .....
   }
```

راه اول، ar را آرایه‌اي با اندازهٔ (یا بزرگي) نامشخص توصیف مي کند. آرایه هم اکنون در تابع اصلي ایجاد شده است . آنچه گذر داده مي‌شود، اشاره‌گری به اولین عنصر از آرایه است ، چون کامپایلر مي داند که عبارت آرایهٔ حاصل، به اشاره‌گر به اولین عنصر آرایه برمي‌گردد، پس ar را مشابه توصیف ar در روش دوم، به اشاره‌گر از نوع float تبدیل مي‌کند. بنابراین هر دو گونه ازنظر نحوهٔ عملکرد، معادل و هم‌ارز یکدیگرند.

به هرحال از نظر وضوح، ممکن است روش اول ترجیح داده شود ، زیرا این روش تأکید مي کند که آنچه باید گذر داده شود آدرس پایه یا آدرس اولین عنصر آرایه است. در روش دوم، راهي وجود ندارد تا بتوان تشخیص داد آیا ar به آغاز آرایه‌ای از نوع float و یا تنها به یك عنصر از نوع float اشاره مي‌کند یا نه.

آرایه‌هایي از اشاره‌گرها

در زبان C مي‌توان آرایه‌اي از اشاره گرها تعریف کرد ؛ یعني آرایه‌اي که عناصر آن اشاره گر باشند. دستور زیر آرایه‌اي 10 عنصري از اشاره گرها را توصیف مي‌کند.

```
int *x[10] ;
```

اینها اشاره‌گرهایي‌اند که مي‌توانند آدرس متغیرهایي از نوع مقادیر صحیح را در خود داشته باشند . به عنوان مثال براي اختصاص دادن آدرس متغیري به نام z به عنصر سوم آرایهٔ مزبور مي‌نویسیم

```
*x[2] = &z ;
```

همین طور براي به دست آوردن مقدار z از دستور [x]** استفاده مي کنیم.

آرایه‌اي از اشاره‌گرها را نیز مي‌توان مشابه آرایه‌هاي معمولي به یك تابع انتقال داد ؛ یعني به سادگي، نام آرایه را بدون اندیس یا زیرنویس آن به عنوان آرگومان تابع قرار مي‌دهیم.

ϖ **مثال 15.8** تابع FF1 مي‌تواند آرایهٔ x را به صورت زیر دریافت کند.

```
void FF1 (int *a[ ])
  {
    int k ;
    for (k=0 ; k<10 ; k+ +)
      printf (" %p" , *a[k]) ;
  }
```

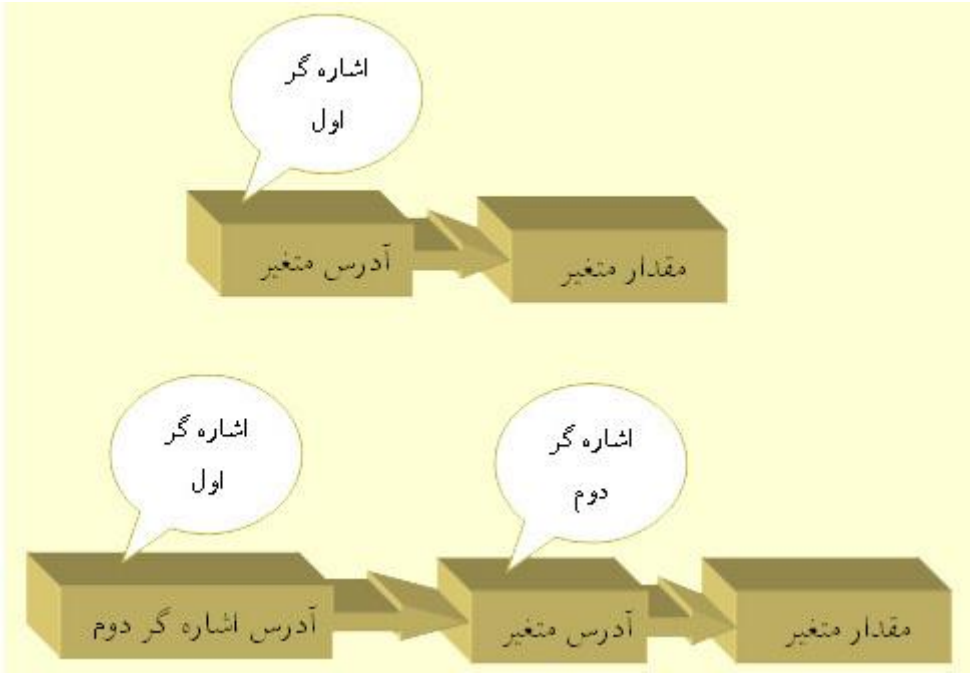
توجه داشته باشید که در این مثال a اشاره‌گری به مقادیر صحیح نیست بلکه اشاره گری به آرایه‌اي از اشاره گرهایي به مقادیر صحیح است. بنابراین نیاز است که پارامتر a آرایه‌اي از اشاره‌گرهایي به مقادیر صحیح، به همان طریق که نشان دادیم، توصیف شود.

آرایه‌هاي اشاره‌گر اغلب براي نگهداري اشاره‌گرهایي به رشته‌ها به کار مي‌روند.

ϖ

اشاره‌گر به اشاره‌گر

گفتیم اگر متغیري آدرس متغیر دیگري را در خود نگه دارد آن را اشاره گر نامند. حال اگر متغیر دوم نیز از نوع اشاره گر باشد، متغیر اول اشاره‌گر به اشاره‌گر است. چنین موقعیتي را اشاره گر به اشاره گر نامند. ممکن است تصور و فهم اشاره گر به اشاره گر ایجاد اشکال کند. شکل 4.8 مفهوم اشاره‌گر به متغیر عادي و اشاره‌گر به اشاره‌گر را روشن مي‌کند.



شكل 4.8 نمايش اشاره‌گر به متغير عادي و اشاره‌گر به اشاره‌گر

همان طور كه ملاحظه مي كنيد، مقدار اشاره گر معمولي، آدرس متغير است اما در مورد اشاره گر به اشاره گر، اولين ا شاره‌گر آدرس اشاره‌گر دوم را دارد كه آن هم به نوبة خود آدرس متغير ديگري را در خود دارد . اين روش مي تواند (به‌صورت تودرتو) تا هرچند بار كه نياز باشد، تكرار گردد. اما در عمل كمتر به بيش از يك بار نياز پيش مي‌آيد و داشتن تصور صحيح از آن نيز براي اغلب برنامه‌نويسان مشكل است.

براي توصيف متغيرهايي از نوع اشاره‌گر به اشاره‌گر بايد دو ستاره در جلوي آن قرار داد. براي مثال توصيف زير به كامپايلر مي‌گويد كه متغير z اشاره‌گر به اشاره‌گری از نوع float است.

```
float **z ;
```

به هرحال بايد توجه داشته باشيد كه z اشاره‌گری به يك مقدار اعشار نيست، بلكه اشاره‌گری به اشاره‌گر است كه اشاره‌گر دوم مي‌تواند آدرس متغيري از نوع float را داشته باشد.

براي دستيابي به مقدار متغير هدف، كه آدرس آن در اشاره‌گر دوم است، بايد اپراتور ستاره را دوباره به كار ببريد، مانند مثال زير.

```
# include <stdio.h>
main ()
  {
    int x , *p , **q ;
    x = 10 ;
    p = &x ;
    q = &p ;
    printf ("%d" , **q) ; / * print the value of x */
  }
```

در اين مثال، p اشاره‌گر به متغير int و q نيز اشاره‌گری به اشاره‌گر توصيف شده است كه ممكن است آدرس متغيري از نوع int را داشته باشد. حال نتيجة اجراي printf اين خواهد شد كه 10 (مقدار متغير x) روي صفحه نمايش نشان داده شود.

يادآوري. آرايه‌اي از اشاره‌گرها نوعي از اشاره‌گر به اشاره‌گر است.

ارسال تابع‌ی به تابع ديگر

در زبان C هنگامي كه تابع‌ی درون تابع ديگري اعلان مي شود آن تابع را تبديل اشاره‌گري به خودش نامند. چنين اشاره‌گرهايي را مي توان به عنوان آرگومان به توابع ديگر فرستاد كه در واقع باعث مي شود تابع متناظر با آنها به تابع ديگري ارسال شود و درون آن بتوان بدان دست يافت. از آنجايي كه آرگومان حقيقي مورد استفاده همواره متغير است، بنابراين در فراخ وانيهاي مختلف تابع دوم مي توان اشاره‌گرهاي گوناگوني (توابع متفاوتي) به آن فرستاد . هر گاه تابعي از طريق آرگومان محلي خود تابعي ديگر را بپذيرد، بايد اعلان آن به گونه‌اي باشد كه مشخص سازد آرگومان محلي مورد نظر اشاره‌گري است به تابع. در ساده‌ترين شكل اين آرگومان را مي‌توان به صورت زير اعلان كرد.

```
data_type (*function_name) () ;
```

كه در آن data_type نوع داده كميت بازگشتي تابع ارسالي است. به دنبال اين اعلان مي‌توان به تابع مزبور با عملگر غيرمستقيم دست يافت.

نتيجه‌گيری

باتوجه به مطالب اين فصل مي‌توان نتايج زير را در مورد اشاره‌گرها بيان كرد.

1. دو عملگر يا اپراتوری كه در اشاره‌گرها استفاده می‌شوند عبارت‌اند از * و &. عملگر & عملگری يكاني است كه آدرس عملوند يا اپراند خود را مشخص مي‌كند. عملگر * نيز عملگری يكاني است كه محتويات آدرس حافظه را مشخص مي‌كند. بنابراين عملگر * مكمل عملگر & است.
2. اعمال متداول روي اشاره‌گرها عبارت‌اند از:

الف) عمل انتساب يا جايگذاري

ب) اعمال محاسباتي شامل جمع، تفريق، + + و - -

ج) مقايسهٔ اشاره‌گرها.
3. اشاره‌گرها داراي ويژگيهاي زيرند.
الف) عمل تخصيص حافظه به‌صورت پويا را امكان‌پذير مي‌كنند.
ب) كار با آرايه‌ها و رشته‌ها را آسان مي‌كنند.
ج) موجب بهبود كارآيي بسياري از توابع مي‌گردند.
د) فراخواني با آدرس را در مورد توابع امكان‌پذير مي‌سازند. درنتيجه برگردان بيش از يك مقدار، از يك تابع، ميسر مي‌گردد.
نكتهٔ 1. كلمهٔ كليدي far براي تعريف اشاره گرهايي به كار می رود كه به ذخيرهٔ آدرسهايي بيش از دو بايت نياز دارند و قبل از نام متغير اشاره‌گر و بعد از تعيين نوع اصلي ذكر مي‌شود، مانند اعلان زير.

```
char far *ptr ;
```

نكتهٔ 2. كلمهٔ كليدي huge براي تعريف اشاره‌گرهايي به كار می‌رود كه آدرسهايي بيش از شانزده بيت را در خود ذخيره مي‌كنند.

خودآزمایی 8

1. تابعي بنويسيد كه با استفاده از اشاره‌گر، طول رشته‌ای را به دست آورد.
2. برنامه‌اي بنويسيد كه يک خط متن از ورودي دريافت كند و تعداد حروف صدادار، حروف بي صدا، فاصله ها و تعداد ارقام را در آن مشخص كند.
3. يكي از ك اربردهاي مهم اشاره گرها در مورد آرايه هاي كاراكتري است . اغلب عمليات روي رشته ها معمولاً با استفاده از عمليات محاسباتي روي اشاره‌گرها انجام مي‌گيرد. تابع زير دو رشته را از لحاظ يكسان بودن با يكديگر مقايسه مي كند. اگر يكسان نبودند true و در غير اين صورت false برمي‌گرداند. در واقع نقش تابع كتابخانه‌اي strcmp را انجام مي‌دهد.
4. تابعي بنويسيد كه با استفاده از اشاره‌گر، دو رشته را به هم متصل كند و رشتهٔ سومي تشكيل دهد.
5. برنامه‌اي بنويسيد كه با استفاده از اشاره گر و روش اختصاص حافظه به صورت پويا، حافظه اي براي آ رايهٔ n عنصري اختصاص دهد و عناصر آرايه را به حافظه بخواند . سپس با فراخوانده شدن يك تابع، عناصر آرايهٔ مزبور با به كارگيري اشاره گر، به صورت صعودي مرتب و نتيجه در تابع اصلي چاپ شود.
6. برنامه‌اي بنويسيد كه عناصر دو ماتريس (آرايهٔ دوبعدي) را به حافظه بخواند و مرجموع آن دو را براساس قانون جمع ماتريسها در آرايهٔ C قرار دهد و نتيجه را به صورت ماتريس چاپ كند.
7. يك راه براي مرتب كردن رشته ها با استفاده از اشاره گرها آن است كه آدرس رشته ها را در آرايه‌اي از اشاره‌گرها قرار دهيم. سپس در مقايسهٔ دو رشته، اگر نياز به جابه جايي آن دو با يكديگر باشد ، آدرس دو رشته را در درون آرايهٔ اشاره گرها كه آدرس رشته ها را دارد با يكديگر عوض كنيم.
8. تابعي بنويسيد كه مقادير دو متغير را با استفاده از اشاره گر تعويض كند.
9. برنامه‌اي بنويسيد كه رشته و كاراكتري را از ورودي بخواند . سپس با فرا خواندن تابع فرعي، تعداد دفعاتي را كه كاراكتر مورد نظر در رشتهٔ مزبور وجود دارد بشمارد و چاپ كند.
10. نحوهٔ اعلان يا توصيف متغيرهای زير در اشاره گرها را شرح دهيد.

```
1. int *p ;
2. int *p[10] ;
3. int (*p)[10] ;
4. int *p(void) ;
5. int p(char *a) ;
6. int *p(char *a) ;
7. int (*p) (char *a) ;
8. int (*p(char *a))[10] ;
9. int p(char(*a)[ ]) ;
10. int p(char *a[ ]) ;
```

```
11. int *p(char a[ ]) ;
12. int *p(char (*a)[ ]) ;
13. int *p(char *a[ ]) ;
14. int (*p)(char (*a)[ ]) ;
15. int *(*p)(char (*a)[ ]) ;
16. int *(*p)(char *a[ ]) ;
17. int (*p[10])(void) ;
18. int (*p[10])(char a) ;
19. int *(*p[10])(char a) ;
20. int *(*p[10])(char *a) ;
```


شبکه آموزشی–پژوهشی مادسیج
با هدف بهبود پیشرفت علمی
و دسترسی راحت به اطلاعات
برای جامعه بزرگ علمی ایران
ایجاد شده است
madsg.com
مادسیج
IRan Education & Research NETwork
(IRERNET)